UMIACS-TR-90-50 April 1990
CS-TR -2449

The Common Prototyping Language

A Module Interconnection Approach

Steve Huseth, Jonathan Krueger, and Aaron Larson
Systems and Research Center
Honeywell

James Purtilo
Institute for Advanced Computer Studies
University of Maryland

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-90-50 April 1990
CS-TR -2449

The Common Prototyping Language
A Module Interconnection Approach

Steve Huseth, Jonathan Krueger, and Aaron Larson
Systems and Research Center
Honeywell

James Purtilo
Institute for Advanced Computer Studies
University of Maryland

Abstract

This report describes a collaborative effort between Honeywell and the University of Maryland towards construction of a Common Prototyping Language. Our goal is to develop a wide-spectrum language offering object-oriented, logic, and functional programming paradigms based on the Common Lisp Object System (CLOS), coupled with a powerful module interconnection capability supplying flexible assembly of existing parts and rapid, distributed reconfigurability. We plan on using an open language design, allowing us to maximize use of widely accepted features and subsystems available within CLOS. Moreover, the construction of a wide spectrum language should provide significant breadth of expression compared to other prototyping languages. Finally, our language will feature a powerful interconnection capability, hence allowing prototypes to be assembled using existing parts.

This collaborative research is sponsored by DARPA/ISTO under the Common Prototyping Language initiative, with oversight provided by Office of Naval Research.

1 INTRODUCTION

DARPA/ISTO's first step towards achieving a general *Common Prototyping System* is to identify language issues within the prototyping process, and to subsequently put forth a prototyping notation to address those issues. Such a *Common Prototyping Language* (CPL) should thereafter support developers as they express an initial design, rapidly formulate specifications for individual components, and then execute those specifications in order to increase their understanding of the application domain. Presumably, the risk that a project will fail is reduced by exposing important properties earlier in the development process.

Our approach to a CPL focuses on *direct execution* of specifications, and on *scale*. In order to understand essential properties of a proposed system, developers must achieve execution rapidly; this implies that they must be able to directly execute their specifications. Moreover, the developer must be able to express and execute specifications for realistic-sized applications, and then organize the observed behavior of this prototype into a useful structure: to support such discourse concerning large applications, it is clear that the CPL must provide features normally found in a module interconnection language.

In a collaborative effort, Honeywell and the University of Maryland are working to provide a wide-spectrum language offering object-oriented, logic, and functional programming paradigms based on the Common Lisp Object System (CLOS), coupled with a powerful module interconnection capability supplying flexible assembly of existing parts and rapid, distributed reconfigurability. Key features to this approach are:

- The open language design maximizes use of widely accepted features and subsystems as available in CLOS.
- The wide spectrum language component has been demonstrated to have significant breadth of expression relative to other comparable languages.
- Our interconnection capabilities will allow prototypes to be assembled using existing components.

As we will show, this research draws heavily upon existing tools which have already been prototyped and demonstrated. Therefore our work is primarily directed towards integration of resources, along with subsequent evaluation and demonstration.

2 TECHNICAL APPROACH

Honeywell has developed a hybrid language that combines functional, logic, and object-oriented paradigms in a language system called ObLog. Originally developed as a common integration language for RADC's Knowledge-Based Software Assistant (KBSA) program, the language is hosted on SUN and Symbolics workstations. ObLog has been demonstrated as a system suitable for programming many traditional and knowledge-based systems, including those originally written in Refine and Socle. ObLog is a "wide spectrum" language due to the breadth of expressiveness it supports.

The thrust of our work is to combine the wide-spectrum language characteristics of ObLog with the module interconnection language Polylith developed at the University of Maryland. While ObLog provides a powerful paradigm for prototyping software components, the Polylith Module Interconnection Language (MIL) provides a formal basis for describing the structure of large sets of software components that have been specified in ObLog or are assembled from existing software written in a variety of traditional languages such as Ada, C, or Pascal. Once an application has been specified and the components interconnected, it may be directly executed. Some elements will be simulated, some will be executed within the ObLog environment, and some components of the application will be implemented by objects drawn from reuse libraries of existing products. All modules may be uniformly integrated through the Polylith runtime across one or more processors.

This approach to a rapid prototyping language differs from other existing language systems for rapid construction of application software. Existing advanced rapid prototyping languages rely heavily on breadth of expressiveness rather than on clarity of interfaces, and consequently are very poor when used by a large team of programmers constructing a single prototype. By joining an advanced very high-level language with a module interconnection language, we will provide both the clarity of inter-programmer interfaces and richness of specification semantics required to rapidly prototype a large application's behavior.

Finally, all semantics specifiable in the language are fully executable and compilable. We have developed techniques resulting in high-performance code execution for the logic and object-oriented aspects of the language. Rapid debugging and instrumentation capabilities are provided to quickly produce operational prototypes that can be used to validate user expectations and form the basis for constructing the production-quality system.

2.1 ObLog LANGUAGE

The Honeywell ObLog language is not a new language but a composite of existing languages that provides a broader range of expressiveness than any of the individual parts. The sublanguages that are included are Common Lisp, CLOS, and LogLisp. Each sublanguage has been joined to the other two using special techniques as shown in Figure 1. The following discussion describes each of these sublanguages and the mechanisms used to integrate them.

CLOS was defined by the ANSI X3J13 Common Lisp Standards Committee and has recently

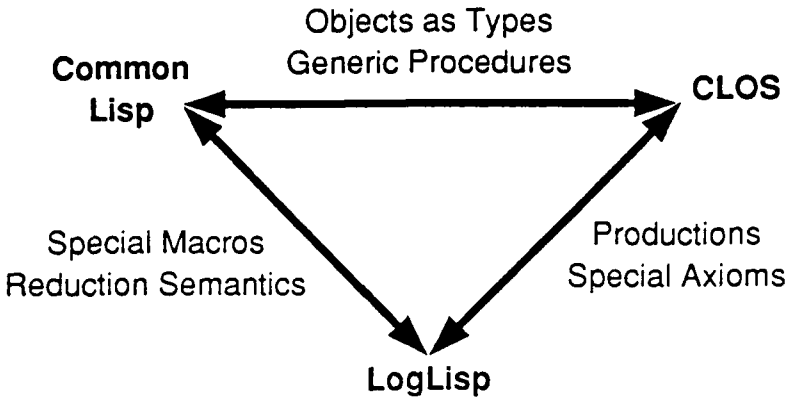

Figure 1: ObLog Language Components

been accepted into Common Lisp Standard. It contains most concepts typically ascribed to object-oriented programming, including:

- Classes, instances, class variables, instance variables, methods
- Multiple inheritance
- Method combination and mechanisms to redefine combination policy
- Metaclasses and mechanisms to redefine combination policy
- Metaclasses and mechanisms for defining new metaclasses
- Mechanisms to dynamically change the class of an instance or all instances of a class

CLOS is joined to Common Lisp by mapping the Common Lisp type space into the space of object classes. Every class has a corresponding type with the same name as the class. Conversely, Common Lisp type specifiers have a class associated with them that has the same name as the type.

LogLisp is a Prolog-like logic system embedded in Common Lisp. It provides an advanced pattern matching and inferencing capability and serves as a query facility through unification on CLOS objects. LogLisp extends Prolog by permitting Lisp reduction to be performed during the Prolog unification. According to the LogLisp unification strategy, rule clauses may contain actions that succeed by returning non-NIL values. Consequently, functions may be used to short-circuit the standard Prolog resolution strategy to improve performance or achieve a unique effect. LogLisp also permits the search strategy to be tailored to be depth first, breadth first, or a combination strategy.

LogLisp is integrated with the object database by mapping clauses into object accessor functions. Clauses are defined that reference the object's instance variables. These clauses permit rules to be constructed to infer additional properties.

In Figure 2, the object class *foo* is defined resulting in the associated clauses being asserted as shown. The <?> notation denotes the handle to the object instance that consists of an unprintable internal data structure pointer. The "!" operator defines the form that follows it to be interpreted as a Lisp action rather than a LogLisp rule.

```
(defclass foo ()
   ((a :initform 10)
    (b :initform 20)))

(make-instance 'foo)
```

Rules asserted of the form:

```
   ((foo-a <?> a 10) <- !(eq (foo-a <?>) 10))
   ((foo-b <?> b 20) <- !(eq (foo-a <?>) 20))

(defpredicate bar
   ((bar ?x) <- (foo-a ?x a 10))
   ((bar ?x) <- (foo-a ?x a 11))
```

Figure 2: ObLog Example

The rule **bar** is defined in the example to demonstrate inferencing on the primitive axioms. The ?x notation is used to denote a free variable in the rule. The **bar** rule defines as true when an object of the class **foo** contains an **a** slot that has the numeric value of 10 or 11.

The integration of the functional and logical paradigms has also provided a forward chaining production mechanism. A production is a LogLisp clause followed by a functional action. The action is invoked on the bound variables of the clause whenever the query is satisfied. This action may result in additional productions being triggered, resulting in cascading effect.

Productions are used to prescribe demon actions or consistency constraints that are to be satisfied by a specification. Optionally, the functional actions can either attempt to correct an inconsistency or simply signal the problem.

2.2 MODULE INTERCONNECTION LANGUAGE

Polylith is a system that supports the interconnection of heterogeneous software modules using an associated hierarchical design. This is done by providing:

- A *module interconnection language* (MIL) to describe modules (in terms of so-called *minimum specifications*), their interfaces and their interconnectivity; and

- A *software bus* to support the run-time communication between modules based on the MIL description.

Abstractly, a Polylith bus exists as an identifiable target to which modules can easily and transparently interface. Transitively, a system's modules may then interface to one another. The intended interconnection between modules is specified by programmers through the Polylith MIL, and is managed at run time by an instance of some Polylith bus. Different instances of buses exist to serve different run-time needs independent of an application's structure, such as instrumentation or debugging. However, while an application's structure is described in this Polylith language, each component module's implementation can be written in whatever application language is appropriate (e.g., Pascal, C, Prolog, or Lisp).

An application system is thought of in terms of independent modules that interact with each other through specified interconnections to form a coherent whole. A *module* embodies a specific functionality and defines a number of *interfaces* or *ports* through which it can communicate with other modules. Furthermore, a set of attributes (e.g., "implementation" information or algebraic specifications) can be associated with each module. In the current Polylith implementation, interfaces correspond to entry points in the code that can be called externally, or to data that are referenced nonlocally. Accordingly, the *interface pattern* specifies either the structure of any transaction across the interface (the order and types of parameters), or the structure and type of the database being accessed nonlocally.

Using the Polylith MIL, the actual interconnectivity (topology) of the system may be specified by instantiating modules (from their generic definitions) and enumerating the bindings between interfaces with compatible interface patterns. The resulting system corresponds directly to a simple attributed graph, where each node represents a module and the directed arcs between them represent bindings between interfaces. This graph analogy is a natural way to view the structure of most large programs, and is an especially useful way for users to view distributed programs.

Consider the various scenarios shown in Figure 3. This shows different ways for binding the instantiation of routine *foo* to one of its uses. Each scenario shown would be established using a different packager and software bus, but all would use precisely the same source implementation. Further, the structural specification would be precisely the same for each scenario. In the first case, each module's implementation resides in the same process space on the same machine – hence the packager would bind the use of *foo* directly to its low-level definition, optimizing the toolbus away.

In the second case, the two implementations are written in different source languages with conflicting representations of the data they share. In this case, the interconnection requires coercion of the data appearing as actual parameters. The packager links the initial *foo* call to a stub appropriate for the first language domain. The interface pattern for this transaction is available from the MIL specification.

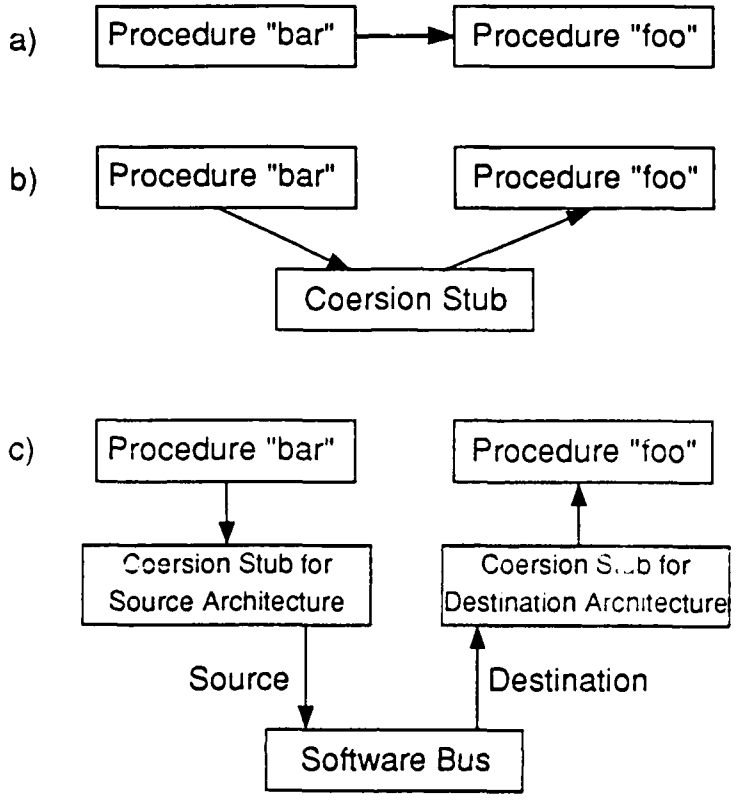

Figure 3: Sample Program Graph

The third scenario is much the same as the second, but accommodates the distribution of the two procedures. Each is assumed to execute on a separate processor, and as a result, the packager not only provides for translation of data, but also for delivery of those data. Hence the stubs generated and linked into the application include calls to the software bus.

There are two important concepts here. The first is the extent to which declarations concerning an application's structure (about the modules and their interconnectivity) are separated from the modules' implementation (in terms of specific programming languages and actual communication media) — with greater independence comes greater flexibility to vary each without negative impact on the other. More importantly, the toolbus encapsulates all information concerning the execution environment, which allows the implementation of individual modules to remain independent of unwanted dependencies upon their context of use. This frees the modular design from considerations of low-level, language-specific details, and also frees the individual implementations from the necessity of knowing the inner workings of other modules or the interconnection media (i.e., they can handle all function calls as if they were in the same process space and written in the same programming language).

The computing community has for years known the value of being able to vary structural specifications independent of module implementations: now, our toolbus organization provides a third

degree of freedom, and that is the freedom to vary the execution environment (e.g., communication medium or host architectures) independent of how the individual modules have been implemented. The software toolbus manages all communication between modules at run time using a Polylith standard data representation protocol. To achieve execution, there must be an instance of a Polylith software bus. The software bus will start up the individual processes and then act as a message delivery system for them. Optionally, a logfile of all transactions done through the bus can be kept for the purposes of instrumentation, analysis, or debugging.

2.3 LANGUAGE and ENVIRONMENT

As languages have become more interactive, the separation between language and the environment has become less distinct. Consequently, the design of the Common Prototyping Language must consider the implications made by the development environment. Our approach of using ObLog as a basis of our language enables us to leverage the environmental aspects that have been defined by the KBSA Framework.

The KBSA Framework is a high-level framework that supports the integration of advanced knowledge-based tools. The environment provides transparent distribution of objects, methods, rules, and demons defined in ObLog across multiple workstations. The existence of the initial environment will provide significant leverage for investigating aspects of the rapid-prototyping language that require environmental support, such as the management of incremental changes to a specification and reuse of existing components.

Management of changes to an emerging specification can be critical to incremental development. Developing a prototype system can be viewed as successive refinement through a sequence of intermediate prototype stages. Each of the prototype stages embodies the decisions and refinements from the previous stage. In developing prototype systems, it is common to perform several development stages and then realize that the path currently being taken is not what is desired. What must be done then is to go back to a previous stage and start again, making a different set of refinements. The two problems involved are 1) recreating the previous stage of the prototype, and 2) making use of the recently obtained knowledge in making the alternative refinements. In each of these cases, it is clear that substantial environmental support for managing rapidly changing configurations will be necessary.

Reuse of existing components also invariably requires environment support resources. A component database, categorization schemes, and selection mechanisms are minimally required. The module interconnection aspect of our language addresses the integration of the component once found, but the environment must be responsible for assisting in initially identifying the component.

2.4 OPERATIONAL SCENARIOS

Scenarios for using the prototyping language are shown in Figures 4 and 5. Each of these figures portrays the flexibility of the module interconnection capability to be used to increase productivity in large prototyping efforts as well as prototyping aspects of existing systems.

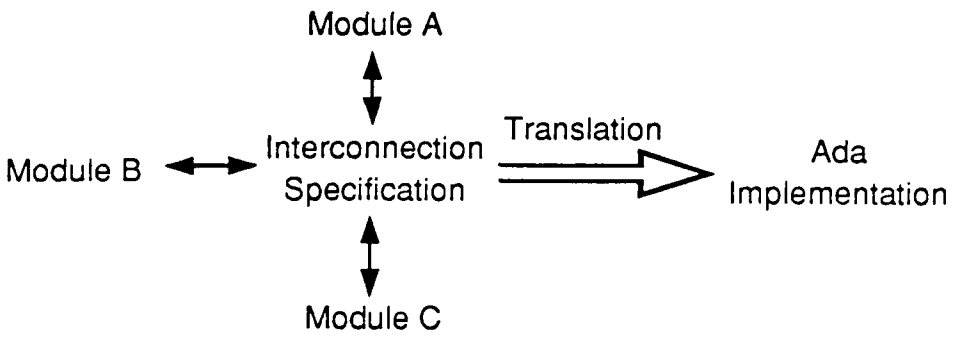

Figure 4: Interconnection of Separately Constructed Modules

In Figure 4, several programmers participate in constructing different parts of a single prototype. Module interconnection descriptions are constructed to define precise interfaces between the different parts. The entire prototype can be integrated using the interface description and a single Ada application generated for the intended target.

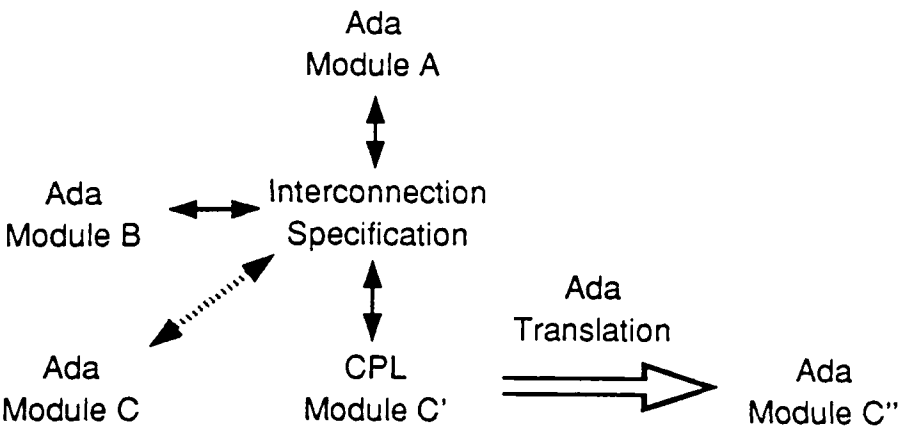

Figure 5: Interconnection of Multi-Language Components

The module interconnection aspect is implementation-language independent. Interfaces can be defined to a variety of systems and hardware platforms. This enables existing software components to be used transparently by a prototyping effort. In Figure 5, an existing Ada application is divided into several modules. These modules may be easily defined along package or procedure boundaries. A module interconnection specification is written for the modules to define their interfaces. It is possible to directly extract these interfaces from the Ada specifications themselves. Once the system has been interconnected, any one of the modules may be removed and replaced

with a CPL module that implements that interface, permitting new aspects of the module to be prototyped and explored.

2.5 INTEROPERABILITY WITH ADA

The language will be closely interoperable with Ada from two perspectives: 1) as discussed earlier, Ada components may be integrated into the evolving prototype using the MIL and software bus, and 2) we anticipate a subset of the language to be translatable into Ada.

Within any prototyping language, mechanisms must exist to evolve from initial prototype into the final production system. In many cases, the final delivery environment may not have a compiler for the prototyping language, and will require translation into an existing language where compilers are available. From our experience in constructing object-oriented systems in Ada, we have developed a strategy for translating high-level object-oriented specifications written in languages such as ObLog into efficient Ada code. Using this technique, an Ada package or "class manager" is generated for each object class; the class manager contains all the data structures and procedures that implement the actual and inherited methods of the class. No fixed method applier such as the Smalltalk "send" is necessary. Each Ada procedure may optionally do its own dynamic type checking based on static analysis of the program.

From our work on LogLisp, we have developed rule compilation techniques to generate efficient procedural code. Using a special set of annotations defined in LogLisp to improve the searching and indexing strategies on the logical assertions, we have been able to show substantial performance enhancements over the unannotated Prolog equivalent. We anticipate additional improvements by expanding on this initial set of annotations.

2.6 ADDITIONAL CONSIDERATIONS

The original DARPA/ISTO request for proposals included several additional issues. The remainder of this section considers the extent to which our approach addresses these issues.

2.6.1 SUPPORT FOR AI

The prototyping language facilitates the development of both expert systems and advanced knowledge-based systems. This capability has been demonstrated using ObLog. The principal objective of the KBSA Framework project is to integrate several diverse expert system tools into a single environment. To date, two such systems have been integrated: the Project Management Assistant (PMA) and the Knowledge-Based Requirements Assistant (KBRA).

The PMA was constructed using the Refine environment. The PMA consists of a number of structured objects representing a project, a set of operations for maintaining the consistency of the objects, and tools for graphically displaying project information such as budgets and

schedules. When the knowledge-base is modified creating a state that matches a triggering condition, a corresponding transformation is applied to achieve consistency.

The integration of this expert system proceeded by constructing a source-to-source translator to extract structured object definitions from Refine source and rewrite them in ObLog. The consistency management tools and their triggering conditions, written in the Refine set-theoretic language, were largely syntactically translated into LogLisp rules and Lisp functions.

The KBRA was produced using the Socle environment. Socle is a frame-based environment that supports a constraint language for specifying functional relationships between frame slots.

The approach taken to integrate KBRA was to develop a metaclass describing the semantics of a Socle frame and instantiate the metaclass in ObLog. This consisted of redefining the mechanism for computing inheritance and for allocating storage for an object's instance variables. The result was an ObLog metaclass that was compatible with other metaclasses in the KBSA Framework yet supported the semantics of Socle. This enabled KBRA to be tightly coupled to other KBSA components without extensive redesign and recoding.

Our experience with integrating PMA and KBRA has demonstrated the ability for ObLog to support a wide range of AI programming styles. The variety of programming styles and tight integration of the programming styles results in a powerful language for specifying prototypes.

2.6.2 SUPPORT FOR DIVERSE ARCHITECTURES

The MIL specifies interfaces between separately constructed modules that logically execute either in a parallel or a distributed fashion. The interconnection approach developed at the University of Maryland provides an ideal mechanism for developing, debugging, and instrumenting distributed and parallel applications.

Our MIL and Polylith toolbus organization provides a technology for interconnecting modules and easily mapping them onto a variety of host configurations. The modules can be implemented in differing languages, with the hosts connected by arbitrary communication media. The set of underlying architectures can be heterogeneous.

This approach will be extended by merging semantic assertions from ObLog with MIL declarations from Polylith, enabling the programmer to specify concurrency via the prototyping language. In designing how these languages are to be merged, we will examine alternative approaches such as Hoore's CSP model, the Ada tasking model, and critical regions as candidate approaches.

2.6.3 SUPPORT OF LARGE SYSTEMS AND RE-USE

The ability to interconnect components developed for a variety of purposes and in a variety of source languages will make a significant impact on the construction of large prototypes and the re-use of existing components. MIL formalizes the interfaces of components that are to be reused

making inclusion into emerging systems easier. MIL can also form the basis of an advanced reusable component query and selection system that will be based on the interface specifications it describes.

Furthermore, as has been described earlier, modules of large systems may be interconnected using MIL and selectively replaced by modules written in CPL. The CPL modules may be extended and expanded to prototype aspects of the large system without reconstructing the entire system in the prototyping language.

2.6.4 SPEED OF DEVELOPMENT

In addition to being easily modifiable and amenable to static analysis for early detection of program bugs, protocols will be defined that permit late binding of monitors and collectors with objects within the system. The monitors and collectors available to a developer will be a function of the programming environment. We take for granted that monitors permitting the graphical display of the dynamic state of objects and functions of object state will be present, and that it will be possible to collect tracing information for object state updates and function calls. We anticipate that the level of detail for collection and display of tracing information will be specified separately, permitting browsing of less detailed tracing information during a run, and then backing up and establishing the state of the environment as specified by the information collection predicates.

BIBLIOGRAPHY

[Balz86] R. Balzer. Living in the Next Generation Operating System, Proceedings of 10th IFIP Congress. Dublin, Ireland, 1-5 September 1986.

[BaPa78] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. Proceedings of the Second Conference European Cooperation on Informatics. Springer-Verlag, (1978).

[Bobr88] D. Bobrow et alia. Common Lisp Object System Specification. Draft submitted to X3J13 (March 1988).

[BCEG85] L. Bouge, W. Choquet, L. Epibourg and M. C. Gaudel. Test sets generation from algebraic specifications using logic programming. Universite de Paris-Sud, LRI, report 240. (November 1985).

[CCHA87] J. Carciofini, T. Colburn, G. Hadden and A. Larson. LogLisp Programming System Users Manual. Honeywell Systems and Research Center, (July 1987).

[CCHA87] J. Carciofini, T. Colburn, G. Hadden and A. Larson. CLF Overview. CLF Project. USC Information Sciences Institute. (November 1985).

[CCHA87] J. Carciofini, T. Colburn, G. Hadden and A. Larson. Knowledge Craft Overview. Carnegie Group, (July 1986).

[DaGa85] J. D. Day and J. D. Gannon. A test oracle based on formal specifications. Proceedings of SOFTFAIR II. San Francisco, CA, (December 1985), pp. 126-130.

[DeLS78] R. A. DeMillo, R. J. Lipton and F. G. Sayward. Hits on test data selection: help for the practicing programmer. **Computer 11**, 4 (April 1978), pp. 34-41.

[DeKr78] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. **IEEE Transactions on Software Engineering**, 2, 2, (June 1978), pp. 80-86.

[GaMH81] J. D. Gannon, P. R. McMullin and R. G. Hamlet. Data Abstraction Implementation, Specifications, and Testing. **TOPLAS**, 3, (July 1981), pp. 211-223.

[GLBC83] C. Green, D. Luckham, D. Balzer, R. Cheatham I, and C. Rich. Report on a Knowledge-Based Software Assistant. Prepared for Rome Air Development Center, Griffiss AFB, New York 13441. (June 1983).

[HaNo86] N. Habermann and D. Notkin. Gandalf: Software Development Environments." **IEEE Trans. on Software Engineering**, 12, 12, (December 1986), pp. 1117-1127.

[Haml77] R. Hamlet. Testing Programs with the Aid of a Compiler. **IEEE Transactions on Software Engineering**, SE-3, (July 1977), pp. 279-289.

[Haye83] B. Hayes-Roth. The Blackboard Architecture: A General Framework for Problem Solving. Stanford University TR HPP-83-30, (May 1983).

[Howd76] W. E. Howden. Reliability of the Path Analysis Testing Strategy. **IEEE Transactions on Software Engineering**, SE-2 (September 1976), pp. 208-215.

[Howd81] W. E. Howden. Completeness Criteria for Testing Elementary Program Functions. Proceedings 5th Int'l Conference on Software Engineering, San Diego, (March 1981), pp. 235-242.

[Huse88] S. Huseth. Framework Support for Knowledge-Based Software Development. Proceedings of Applications of AI6, Orlando, (April 1988).

[KaFP88] G. Kaiser, P. Feiler, S. Popovick. Intelligent Assistance for Software Development and Maintenance. Columbia University, (June 1987).

[LaKo83] J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. **IEEE Transactions on Software Engineering**, SE-9, (May 1983), pp. 347-354.

[Luqi89a] Luqi. Automated Prototyping Data and Knowledge Translation. To appear in Journal of Data and Knowledge Engineering, (1989).

[Luqi88] Luqi. Knowledge-Based Support for Rapid Prototyping. IEEE Expert, (November 1988).

[Luqi89b] Luqi. Rapid Prototyping Language for Expert Systems. IEEE Expert, (May 1989).

[LuBY88] Luqi, V. Berzins, R. Yeh. A Prototyping Language for Real-Time Software. **IEEE Transactions on Software Engineering**, (October 1988).

[LuBe89a] Luqi, V. Berzins. Issues in Language Support for Rapid Prototyping. Naval Postgraduate School, TR 52-89-026, (1989).

[LuBe89b] Luqi, V. Berzins. Rapid Prototyping Languages in Computer Aided Software Engineering. Naval Postgraduate School, TR 52-89-025, (1989).

[Oster86] L. Osterweil. Software Processes are Software Too. Proceedings of the 9th Conference on Softwar Engineering, (March 1986).

[McGa81] P. R. McMullin and J. D. Gannon. Evaluating a Data Abstraction Testing System Based on Formal Specifications. **Journal of Systems and Software Science** 2, 2, (1981), pp. 177-186.

[McGa83] P. R. McMullin and J. D. Gannon. Combining Testing with Formal Specifications: A Case Study. **IEEE Transactions on Software Engineering**, SE-9, (May 1983), pp. 328-334.

[NeWL81] J. Nestor, W. Wulf, D. Lamb. IDL-Interface Description Language. Carnegie-Mellon University TR CMU-CS-81-139, (August 1981).

[PSSS85] H. Pesch, H. Schaller, P. Schnupp and A. Spirk. Test Case Generation Using Prolog. Proceedings 8th Int'l Conference on Software Engineering, London, (August 1985), pp. 252-258.

[Purt85] J. Purtilo. Polylith: An Environment to Support Management of Tool Interfaces. Proceedings of ACM SIGPLAN Symposium on Language Issues in Programming Environments, (July 1985), pp. 12-18.

[Purt86] J. Purtilo. Applications of a Software Interconnection System in Mathematical Problem Solving Environments. Proceedings of Symposium on Symbolic and Algebraic Computation, (July 1986), pp. 16-23.

[Purt88] J. Purtilo, D. Reed and D. Grunwald. Environments for parallel algorithms. **Journal of Parallel and Distributed Computing**, vol. 5, (1988), pp. 421-437.

[RaWe82] S. Rapps and E. J. Weyuker. Data Flow Analysis Techniques for Test Data Selection. Proceedings 6th Int'l Conference on Software Engineering, Tokyo, (September 1982), pp. 272-278.

[RoSi82] J. Robinson, E. Sibert. LogLisp: Motivation, Design and Implementation. In K. L. Clark and S. Tarnlund (ed.), Logic Programming, Academic Press, 1982, pp. 299-313.

[Smit85] D. Smith, et alia. Research on Knowledge-Base Software Environments as Kestrel Institute. **IEEE Transactions on Software Engineering**, (November 1985).

[Swee85] R. E. Sweet, The Mesa Programming Environment. Proceedings of the ACM SIGPLAN Symposium on Programming Issues in Programming Environments, (June 1985), pp. 216-229.

[Tayl87] R. Taylor, et alia. Next Generation Software Environments: Principles, Problems, and Research Directions. University of California at Irvine, TR 87-16 (1987).

[ViKi90] D. Vines, T. King. Gaia: An Object-Oriented Framework for an Ada Environment. **IEEE Software Engineering with Ada**.